LOW LEVEL DESIGN (LLD) DOCUMENT

Order Management Service - Swiggy-like Delivery System

1. SYSTEM OVERVIEW

1.1 Purpose

Design a scalable, maintainable order management system for food and grocery delivery with comprehensive order lifecycle management.

1.2 Key Requirements

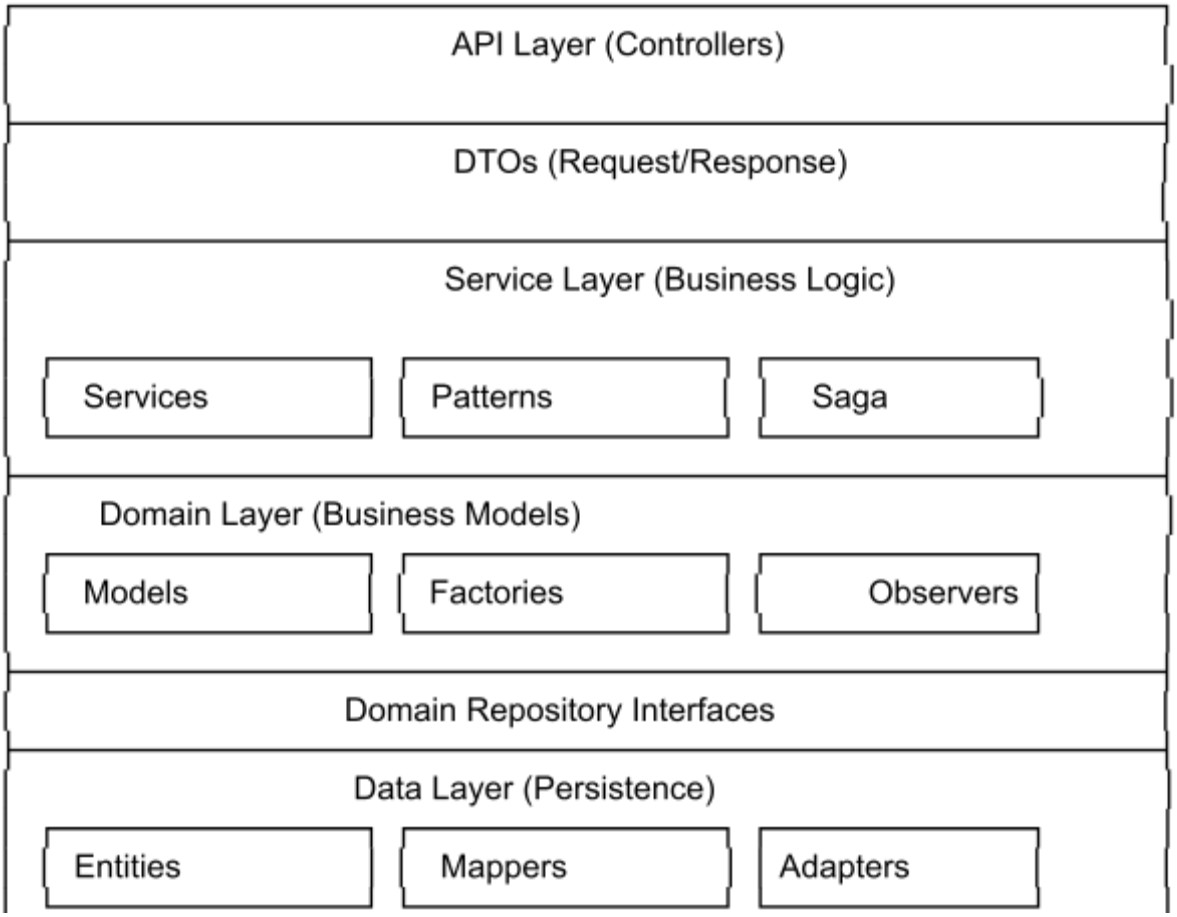
- Order Lifecycle: CREATED → ASSIGNED → PICKED → ON_THE_WAY → DELIVERED → CANCELLED
- City-based routing with multiple assignment strategies
- Idempotent operations for reliability
- Real-time status updates with event notifications
- Pagination & filtering for order search
- Delivery partner management with capacity control

1.3 Non-Functional Requirements

- Availability: 99.9% uptime
- Scalability: Handle 10K orders/minute peak
- Latency: < 200ms for order creation, < 100ms for reads
- Data Consistency: Eventual consistency with Saga pattern
- Testability: High test coverage with clean boundaries

2. ARCHITECTURE DESIGN

2.1 Clean Architecture Layers



2.2 Design Pattern Map

Design Pattern Usage	
Pattern	**Purpose**
Factory	**Complex domain object creation**
Strategy	**Delivery assignment algorithms**
Observer	**Order status change notifications**
Template	**Order processing workflows**
Chain	**Order validation pipeline**
Saga	**Distributed transaction management**
Adapter	**Repository implementations**
Builder	**Immutable object construction**
Repository	**Data access abstraction**

3. DOMAIN MODEL DESIGN

3.1 Core Entities

3.1.1 Order Domain Model

```
/**
 * Rich Domain Model - Pure business logic, no JPA dependencies
 * Encapsulates order lifecycle and business rules
 */
public class Order {
    // Identity
    private String id;                  // UUID
    private String orderId;             // External ID: OMS-2024-001
    private String customerId;
    private String customerName;
    private String restaurantId;
    private String restaurantName;

    // State
    private OrderType orderType;        // FOOD, GROCERY, PHARMACY
    private OrderStatus status;         // CREATED → DELIVERED
    private PaymentStatus paymentStatus; // PENDING → COMPLETED

    // Location
    private Address deliveryAddress;
    private Address restaurantAddress;

    // Items & Pricing
    private List<OrderItem> items;      // Value objects
    private BigDecimal totalAmount;
    private BigDecimal deliveryCharge;
    private BigDecimal taxAmount;
    private BigDecimal grandTotal;

    // Delivery
    private String deliveryPartnerId;
    private Integer estimatedDeliveryMinutes;
    private Integer actualDeliveryMinutes;

    // Metadata
    private String specialInstructions;
    private String cancellationReason;

    // Timestamps
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;
    private LocalDateTime assignedAt;
    private LocalDateTime pickedAt;
    private LocalDateTime deliveredAt;
```

```
    private LocalDateTime cancelledAt;

    // Business Methods
    public boolean canBeCancelled() { ... }
    public boolean canBeAssigned() { ... }
    public void calculateGrandTotal() { ... }
    public void assignToDeliveryPartner(String partnerId) { ... }
    public void updateStatus(OrderStatus newStatus) { ... }
    public void cancel(String reason) { ... }

    // State Transition Validation
    private void validateStatusTransition(OrderStatus newStatus) { ... }
}
```

3.1.2 DeliveryPartner Domain Model

```
/**
 * Delivery Partner with capacity management
 */
public class DeliveryPartner {
    private String id;
    private String name;
    private String phoneNumber;
    private String email;
    private DeliveryPartnerStatus status;    // AVAILABLE, BUSY, OFFLINE

    // Location
    private Address currentLocation;

    // Vehicle
    private String vehicleType;              // BIKE, CAR, SCOOTER
    private String vehicleNumber;

    // Capacity Management
    private Integer assignedOrderCount;      // Current load
    private Boolean isActive;

    // Business Methods
    public boolean isAvailable() { ... }
    public boolean canAcceptOrder() { ... }  // Max 3 orders
    public void assignOrder() { ... }        // Update count & status
    public void releaseOrder() { ... }       // Decrement count
}
```

3.2 Value Objects

```
/**
 * Immutable value objects - No identity, compared by value
 */
public class Address {
    private String streetAddress;
    private String city;                    // Key for routing
    private String state;
    private String postalCode;
    private String country;
    private Double latitude;                // For distance calculation
    private Double longitude;

    public boolean isInSameCity(Address other) { ... }
}
public class OrderItem {
    private String itemId;
    private String itemName;
    private Integer quantity;
    private BigDecimal unitPrice;

    public BigDecimal getTotalPrice() { ... }
}
```

3.3 Enums (Domain Concepts)

```
// Order State Machine
public enum OrderStatus {
    CREATED,        // Order created, waiting for assignment
    ASSIGNED,       // Assigned to delivery partner
    PICKED,         // Picked up from restaurant
    ON_THE_WAY,     // Out for delivery
    DELIVERED,      // Successfully delivered
    CANCELLED,      // Order cancelled
    FAILED          // Delivery failed
}
public enum OrderType {
    FOOD, GROCERY, PHARMACY, OTHER
}
public enum PaymentStatus {
    PENDING, COMPLETED, FAILED, REFUNDED
}
public enum DeliveryPartnerStatus {
    AVAILABLE, BUSY, OFFLINE, BREAK
}
```

4. SERVICE LAYER DESIGN

4.1 Core Services

4.1.1 OrderService

```
@Service
@Transactional
@Slf4j
public class OrderService {

    // Dependencies following Dependency Inversion
    private final OrderRepository orderRepository;      // Domain interface
    private final OrderFactory orderFactory;            // Factory pattern
    private final OrderValidationChain validationChain; // Chain of Responsibility
    private final OrderStatusChangeObserver observer;   // Observer pattern
    private final OrderCreationSaga orderCreationSaga;  // Saga pattern

    /**
     * Order Creation Flow:
     * 1. Factory creates domain object
     * 2. Chain validates business rules
     * 3. Saga executes distributed transaction
     * 4. Repository persists
     * 5. Observer notifies interested parties
     */
    public Order createOrder(CreateOrderRequest request) {
        // 1. Factory pattern for complex creation
        Order order = orderFactory.createOrder(
            request.getCustomerId(),
            request.getCustomerName(),
            // ... other parameters
        );

        // 2. Chain of Responsibility for validation
        if (!validationChain.validateOrder(order)) {
            throw new ValidationException("Order validation failed");
        }

        // 3. Saga pattern for distributed transaction
        orderCreationSaga.execute(order);

        // 4. Repository pattern for persistence
        Order savedOrder = orderRepository.save(order);

        // 5. Observer pattern for notifications
        observer.notifyOrderCreated(savedOrder);

        return savedOrder;
    }
```

```
    /**
     * Status Update with State Pattern behavior
     */
    public Order updateOrderStatus(String orderId, UpdateOrderStatusRequest request) {
        Order order = getOrder(orderId);
        OrderStatus oldStatus = order.getStatus();

        // Domain model encapsulates state transition logic
        order.updateStatus(request.getStatus());

        Order updatedOrder = orderRepository.save(order);

        // Observer pattern for event propagation
        observer.notifyOrderStatusChanged(updatedOrder, oldStatus, request.getStatus());

        return updatedOrder;
    }
}
```

4.1.2 DeliveryPartnerService

```
@Service
@Transactional
@Slf4j
public class DeliveryPartnerService {

    private final DeliveryPartnerRepository partnerRepository;
    private final AssignmentStrategyFactory strategyFactory; // Strategy pattern

    /**
     * Assignment Flow using Strategy Pattern:
     * 1. Get available partners
     * 2. Select strategy based on context
     * 3. Strategy selects optimal partner
     * 4. Update domain objects
     */
    public DeliveryPartner assignDeliveryPartnerToOrder(String orderId, String partnerId) {
        Order order = orderService.getOrder(orderId);
        DeliveryPartner partner = getDeliveryPartner(partnerId);

        // Domain model validation
        if (!order.canBeAssigned()) {
            throw new InvalidOrderStateException(...);
        }
        if (!partner.canAcceptOrder()) {
            throw new InvalidOrderStateException(...);
        }

        // Domain model business logic
        order.assignToDeliveryPartner(partnerId);
```

```
        partner.assignOrder();

        // Repository saves
        partnerRepository.save(partner);
        orderService.updateOrderStatus(orderId,
            UpdateOrderStatusRequest.builder().status(OrderStatus.ASSIGNED).build());

        return partner;
    }

    /**
     * Find optimal partner using Strategy pattern
     */
    public DeliveryPartner findOptimalPartner(Order order, String strategyName) {
        List<DeliveryPartner> availablePartners =
            partnerRepository.findAvailableDeliveryPartnersByCity(
                order.getRestaurantAddress().getCity());

        // Strategy pattern for algorithm selection
        OrderAssignmentStrategy strategy = strategyFactory.getStrategy(strategyName);
        return strategy.selectDeliveryPartner(order, availablePartners);
    }
}
```

4.2 Design Pattern Implementations

4.2.1 Factory Pattern

```
@Component
public class OrderFactory {

    /**
     * Factory Method: Encapsulates complex creation logic
     * - ID generation
     * - Default values
     * - Business rule validation during creation
     */
    public Order createOrder(String customerId, String customerName, ...) {
        String orderId = generateOrderId();

        Order order = Order.builder()
            .id(UUID.randomUUID().toString())
            .orderId(orderId)
            .customerId(customerId)
            .customerName(customerName)
            .status(OrderStatus.CREATED)          // Default state
            .paymentStatus(PaymentStatus.PENDING) // Default payment
            .createdAt(LocalDateTime.now())
            .updatedAt(LocalDateTime.now())
            .build();
```

```
        // Business logic during creation
        order.getItems().addAll(items);
        order.calculateGrandTotal();

        return order;
    }

// Factory Method variant for different order types
    public Order createExpressOrder(...) {
        Order order = createOrder(...);
        order.setDeliveryCharge(BigDecimal.valueOf(49.99)); // Express premium
        order.setEstimatedDeliveryMinutes(20);              // Faster delivery
        return order;
    }
}
```

4.2.2 Strategy Pattern

```
/**
 * Strategy Interface
 */
public interface OrderAssignmentStrategy {
    DeliveryPartner selectDeliveryPartner(Order order, List<DeliveryPartner>
availablePartners);
    String getStrategyName();
}

/**
 * Concrete Strategy 1: City-based assignment
 */
@Component
public class CityBasedAssignmentStrategy implements OrderAssignmentStrategy {

    @Override
    public DeliveryPartner selectDeliveryPartner(Order order, List<DeliveryPartner>
partners) {
        // 1. Filter by same city
        // 2. Filter by capacity
        // 3. Sort by load (least loaded first)
        // 4. Return optimal partner
        return partners.stream()
            .filter(p -> p.getCurrentLocation().isInSameCity(order.getRestaurantAddress()))
            .filter(DeliveryPartner::canAcceptOrder)
            .sorted(Comparator.comparing(DeliveryPartner::getAssignedOrderCount))
            .findFirst()
            .orElse(null);
    }

    @Override public String getStrategyName() { return "CITY_BASED"; }
}
```

```
/**
 * Concrete Strategy 2: Nearest-first assignment
 */
@Component
public class NearestFirstAssignmentStrategy implements OrderAssignmentStrategy {

    @Override
    public DeliveryPartner selectDeliveryPartner(Order order, List<DeliveryPartner>
partners) {
        // 1. Calculate distance for each partner
        // 2. Filter by capacity
        // 3. Return nearest available partner
        return partners.stream()
            .filter(DeliveryPartner::canAcceptOrder)
            .min(Comparator.comparingDouble(p ->
                calculateDistance(p.getCurrentLocation(), order.getRestaurantAddress())))
            .orElse(null);
    }
}

/**
 * Strategy Factory - Selects strategy at runtime
 */
@Component
public class AssignmentStrategyFactory {
    private final Map<String, OrderAssignmentStrategy> strategies = new HashMap<>();

    public AssignmentStrategyFactory(CityBasedAssignmentStrategy cityStrategy,
                                     NearestFirstAssignmentStrategy nearestStrategy) {
        strategies.put(cityStrategy.getStrategyName(), cityStrategy);
        strategies.put(nearestStrategy.getStrategyName(), nearestStrategy);
    }

    public OrderAssignmentStrategy getStrategy(String name) {
        return strategies.getOrDefault(name, strategies.get("CITY_BASED"));
    }
}
```

4.2.3 Observer Pattern

```
/**
 * Observer Interface
 */
public interface OrderObserver {
    void onOrderStatusChanged(Order order, OrderStatus oldStatus, OrderStatus
newStatus);
    void onOrderCreated(Order order);
    void onOrderCancelled(Order order);
}
```

```
/**
 * Subject/Observable
 */
@Component
public class OrderStatusChangeObserver {
    private final List<OrderObserver> observers = new ArrayList<>();

    public void registerObserver(OrderObserver observer) {
        observers.add(observer);
    }

    public void notifyOrderStatusChanged(Order order, OrderStatus oldStatus,
OrderStatus newStatus) {
        for (OrderObserver observer : observers) {
            observer.onOrderStatusChanged(order, oldStatus, newStatus);
        }
    }
}
/**
 * Concrete Observer 1: Logging
 */
@Component
public class LoggingOrderObserver implements OrderObserver {
    @Override
    public void onOrderStatusChanged(Order order, OrderStatus oldStatus, OrderStatus
newStatus) {
        log.info("Order {}: {} → {}", order.getOrderId(), oldStatus, newStatus);
    }
}
/**
 * Concrete Observer 2: Notification
 */
@Component
public class NotificationOrderObserver implements OrderObserver {
    @Override
    public void onOrderStatusChanged(Order order, OrderStatus oldStatus, OrderStatus
newStatus) {
        if (newStatus == OrderStatus.DELIVERED) {
            notificationService.sendDeliveryNotification(order.getCustomerId());
        }
    }
}
```

4.2.4 Template Method Pattern

```
/**
 * Template defining order processing algorithm
 */
public abstract class OrderProcessingTemplate {

    // Template Method - defines algorithm skeleton
```

```
    public final void processOrder(Order order) {
        validateOrder(order);
        processPayment(order);
        assignDeliveryPartner(order);
        notifyRestaurant(order);
        updateOrderStatus(order);
        completeOrder(order);
    }

    // Abstract steps - subclasses must implement
    protected abstract void validateOrder(Order order);
    protected abstract void processPayment(Order order);
    protected abstract void assignDeliveryPartner(Order order);
    protected abstract void notifyRestaurant(Order order);

    // Concrete steps with default implementation
    protected void updateOrderStatus(Order order) {
        log.info("Updating order status");
    }

    protected void completeOrder(Order order) {
        log.info("Order processing complete");
    }

    // Hook method - subclasses can override
    protected boolean needsSpecialHandling(Order order) {
        return false;
    }
}
/**
 * Concrete Template 1: Standard Order Processing
 */
@Component
public class StandardOrderProcessor extends OrderProcessingTemplate {
    @Override
    protected void validateOrder(Order order) {
        // Standard validation logic
        if (order.getItems().isEmpty()) {
            throw new ValidationException("Order must have items");
        }
    }

    @Override
    protected void processPayment(Order order) {
        // Standard payment processing
        paymentService.processStandardPayment(order);
    }
    // ... other implementations
}
```

```
/**
 * Concrete Template 2: Express Order Processing
 */
@Component
public class ExpressOrderProcessor extends OrderProcessingTemplate {
    @Override
    protected void validateOrder(Order order) {
        // Express-specific validation
        super.validateOrder(order);
        if (order.getEstimatedDeliveryMinutes() > 30) {
            throw new ValidationException("Express orders must deliver in < 30 mins");
        }
    }

    @Override
    protected void processPayment(Order order) {
        // Express payment - immediate processing
        paymentService.processExpressPayment(order);
    }

    @Override
    protected boolean needsSpecialHandling(Order order) {
        return true; // Express always needs special handling
    }
}
```

4.2.5 Chain of Responsibility Pattern

```
/**
 * Handler Interface
 */
public abstract class ValidationHandler {
    private ValidationHandler nextHandler;

    public ValidationHandler setNext(ValidationHandler handler) {
        this.nextHandler = handler;
        return handler;
    }

    public abstract boolean validate(Order order);

    protected boolean validateNext(Order order) {
        if (nextHandler == null) return true;
        return nextHandler.validate(order);
    }
}
/**
 * Concrete Handler 1: Customer Validation
 */
@Component
public class CustomerValidationHandler extends ValidationHandler {
```

```
    @Override
    public boolean validate(Order order) {
        if (order.getCustomerId() == null) {
            log.error("Customer ID required");
            return false;
        }
        return validateNext(order);
    }
}
/**
 * Concrete Handler 2: Restaurant Validation
 */
@Component
public class RestaurantValidationHandler extends ValidationHandler {
    @Override
    public boolean validate(Order order) {
        if (order.getRestaurantId() == null) {
            log.error("Restaurant ID required");
            return false;
        }
        return validateNext(order);
    }
}

/**
 * Chain Builder
 */
@Component
public class OrderValidationChain {
    private ValidationHandler chain;

    @PostConstruct
    public void initializeChain() {
        // Build chain: Customer → Restaurant → Items
        chain = customerValidationHandler;
        chain.setNext(restaurantValidationHandler)
             .setNext(itemsValidationHandler);
    }

    public boolean validateOrder(Order order) {
        return chain.validate(order);
    }
}
```

4.2.6 Saga Pattern

```
/**
 * Saga Step Interface
 */
public interface OrderSagaStep {
    void process(Order order);
    void rollback(Order order);
    String getStepName();
}
/**
 * Saga Coordinator
 */
public abstract class OrderSaga {
    private final List<OrderSagaStep> steps = new ArrayList<>();
    private final List<OrderSagaStep> completedSteps = new ArrayList<>();

    protected void addStep(OrderSagaStep step) {
        steps.add(step);
    }

    public void execute(Order order) {
        try {
            for (OrderSagaStep step : steps) {
                step.process(order);
                completedSteps.add(step);
            }
        } catch (Exception e) {
            rollback(order);
            throw new SagaExecutionException("Saga failed", e);
        }
    }

    private void rollback(Order order) {
        // Rollback in reverse order
        for (int i = completedSteps.size() - 1; i >= 0; i--) {
            try {
                completedSteps.get(i).rollback(order);
            } catch (Exception e) {
                log.error("Rollback failed for step: {}", completedSteps.get(i).getStepName(),
e);
            }
        }
    }
}
/**
 * Concrete Saga for Order Creation
 */
@Component
public class OrderCreationSaga extends OrderSaga {
```

```
    public OrderCreationSaga(PaymentProcessingStep paymentStep,
                             RestaurantNotificationStep restaurantStep,
                             DeliveryAssignmentStep deliveryStep) {
        addStep(paymentStep);
        addStep(restaurantStep);
        addStep(deliveryStep);
    }
}

/**
 * Saga Step Implementations
 */
@Component
public class PaymentProcessingStep implements OrderSagaStep {
    @Override
    public void process(Order order) {
        paymentService.process(order);
    }

    @Override
    public void rollback(Order order) {
        paymentService.refund(order);
    }

    @Override public String getStepName() { return "PAYMENT_PROCESSING"; }
}
```

5. DATA LAYER DESIGN

5.1 Repository Pattern Implementation

5.1.1 Domain Repository Interface

```
/**
 * Domain Repository - Domain layer interface
 * No Spring/JPA dependencies
 */
public interface OrderRepository {
    Order save(Order order);
    Optional<Order> findById(String id);
    Optional<Order> findByOrderId(String orderId);
    List<Order> findByCustomerId(String customerId);
    List<Order> findByStatus(OrderStatus status);
    List<Order> findByCityAndStatus(String city, OrderStatus status);
    boolean existsByOrderId(String orderId);
    void deleteById(String id);
}
```

5.1.2 Adapter Pattern Implementation

```
/**
 * Adapter: Implements domain interface using JPA
 */
@Component
public class OrderRepositoryImpl implements OrderRepository {

    private final OrderJpaRepository jpaRepository;  // Spring Data JPA
    private final OrderEntityMapper mapper;          // Entity ↔ Domain mapper

    @Override
    public Order save(Order order) {
        OrderEntity entity = mapper.toEntity(order);
        OrderEntity savedEntity = jpaRepository.save(entity);
        return mapper.toDomain(savedEntity);
    }

    @Override
    public Optional<Order> findByOrderId(String orderId) {
        return jpaRepository.findByOrderId(orderId)
            .map(mapper::toDomain);
    }

    // Other methods follow same pattern
}
```

5.1.3 JPA Repository (Infrastructure)

```
/**
 * Spring Data JPA Repository
 * Infrastructure concern only
 */
@Repository
public interface OrderJpaRepository extends JpaRepository<OrderEntity, String> {

    Optional<OrderEntity> findByOrderId(String orderId);

    Page<OrderEntity> findByCustomerId(String customerId, Pageable pageable);

    Page<OrderEntity> findByStatus(OrderStatus status, Pageable pageable);

    @Query("SELECT o FROM OrderEntity o WHERE o.deliveryAddress.city = :city AND
o.status = :status")
    Page<OrderEntity> findByCityAndStatus(@Param("city") String city,
                                          @Param("status") OrderStatus status,
                                          Pageable pageable);
}
```

5.2 Entity Design

```
/**
 * JPA Entity - Pure persistence concern
 */
@Entity
@Table(name = "orders", indexes = {
    @Index(name = "idx_order_status", columnList = "status"),
    @Index(name = "idx_order_customer_id", columnList = "customerId"),
    @Index(name = "idx_order_restaurant_id", columnList = "restaurantId"),
    @Index(name = "idx_order_created_at", columnList = "createdAt")
})
public class OrderEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private String id;

    @Column(nullable = false, unique = true)
    private String orderId;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    private OrderStatus status;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "city", column = @Column(name = "delivery_city"))
    })
```

```
    private AddressEntity deliveryAddress;

    @ElementCollection
    @CollectionTable(name = "order_items", joinColumns = @JoinColumn(name =
"order_id"))
    private List<OrderItemEntity> items;

    // Audit fields
    @CreationTimestamp
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime updatedAt;

    // No business logic here - just data persistence
}
```

6. API LAYER DESIGN

6.1 REST Controllers

6.1.1 OrderController

```
@RestController
@RequestMapping("/api/v1/orders")
@Tag(name = "Orders", description = "Order management endpoints")
@Validated
@Slf4j
public class OrderController {

    private final OrderService orderService;
    private final OrderMapper orderMapper;

    /**
     * Create Order
     * POST /api/v1/orders
     */
    @PostMapping
    @Operation(summary = "Create a new order")
    @ResponseStatus(HttpStatus.CREATED)
    public ResponseEntity<OrderResponse> createOrder(
            @Valid @RequestBody CreateOrderRequest request,
            @RequestHeader(value = "Idempotency-Key", required = false) String
idempotencyKey) {

        log.info("Creating order for customer: {}", request.getCustomerId());

        // Idempotency check
        if (idempotencyKey != null && orderService.isDuplicateRequest(idempotencyKey)) {
            return ResponseEntity.status(HttpStatus.CONFLICT).build();
        }

        // Service layer call
        Order order = orderService.createOrder(request);

        // Map to response DTO
        OrderResponse response = orderMapper.toResponse(order);

        return ResponseEntity.status(HttpStatus.CREATED)
            .header("Location", "/api/v1/orders/" + order.getOrderId())
            .body(response);
    }

    /**
     * Get Order by ID
     * GET /api/v1/orders/{orderId}
```

```
     */
    @GetMapping("/{orderId}")
    @Operation(summary = "Get order by ID")
    public ResponseEntity<OrderResponse> getOrder(@PathVariable String orderId) {
        Order order = orderService.getOrder(orderId);
        OrderResponse response = orderMapper.toResponse(order);
        return ResponseEntity.ok(response);
    }

    /**
     * Update Order Status
     * PUT /api/v1/orders/{orderId}/status
     */
    @PutMapping("/{orderId}/status")
    @Operation(summary = "Update order status")
    public ResponseEntity<OrderResponse> updateOrderStatus(
            @PathVariable String orderId,
            @Valid @RequestBody UpdateOrderStatusRequest request) {

        Order order = orderService.updateOrderStatus(orderId, request);
        OrderResponse response = orderMapper.toResponse(order);
        return ResponseEntity.ok(response);
    }

    /**
     * Search Orders with Pagination
     * GET /api/v1/orders
     */
    @GetMapping
    @Operation(summary = "Search orders with filters")
    public ResponseEntity<PagedResponse<OrderResponse>> searchOrders(
            @RequestParam(required = false) String customerId,
            @RequestParam(required = false) String restaurantId,
            @RequestParam(required = false) OrderStatus status,
            @RequestParam(required = false) String city,
            @PageableDefault(size = 20, sort = "createdAt", direction = Sort.Direction.DESC)
            Pageable pageable) {

        Page<Order> orders = orderService.searchOrders(customerId, restaurantId, status,
city, pageable);
        PagedResponse<OrderResponse> response =
orderMapper.toPagedResponse(orders);
        return ResponseEntity.ok(response);
    }
}
```

6.1.2 DeliveryPartnerController

```
@RestController
@RequestMapping("/api/v1/delivery-partners")
@Tag(name = "Delivery Partners", description = "Delivery partner management")
```

```
@Validated
@Slf4j
public class DeliveryPartnerController {

    private final DeliveryPartnerService partnerService;
    private final DeliveryPartnerMapper partnerMapper;

    /**
     * Assign Partner to Order
     * PUT /api/v1/delivery-partners/{partnerId}/assign/{orderId}
     */
    @PutMapping("/{partnerId}/assign/{orderId}")
    @Operation(summary = "Assign delivery partner to order")
    public ResponseEntity<DeliveryPartnerResponse> assignToOrder(
            @PathVariable String partnerId,
            @PathVariable String orderId,
            @Valid @RequestBody AssignDeliveryPartnerRequest request) {

        DeliveryPartner partner = partnerService.assignDeliveryPartnerToOrder(orderId,
partnerId);
        DeliveryPartnerResponse response = partnerMapper.toResponse(partner);
        return ResponseEntity.ok(response);
    }

    /**
     * Get Available Partners by City
     * GET /api/v1/delivery-partners/available/{city}
     */
    @GetMapping("/available/{city}")
    @Operation(summary = "Get available delivery partners by city")
    public ResponseEntity<List<DeliveryPartnerResponse>> getAvailablePartners(
            @PathVariable String city,
            @RequestParam(defaultValue = "CITY_BASED") String strategy) {

        List<DeliveryPartner> partners = partnerService.findAvailablePartnersByCity(city,
strategy);
        List<DeliveryPartnerResponse> response = partners.stream()
            .map(partnerMapper::toResponse)
            .collect(Collectors.toList());
        return ResponseEntity.ok(response);
    }
}
```

6.2 DTO Design

```
/**
 * Request DTO with validation
 */
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Schema(description = "Request to create a new order")
public class CreateOrderRequest {

    @NotBlank(message = "Customer ID is required")
    @Schema(description = "Unique identifier of the customer", example = "cust_12345")
    private String customerId;

    @NotBlank(message = "Customer name is required")
    @Schema(description = "Name of the customer", example = "John Doe")
    private String customerName;

    @NotNull(message = "Order type is required")
    @Schema(description = "Type of the order", example = "FOOD")
    private OrderType orderType;

    @NotNull(message = "Delivery address is required")
    @Valid
    private Address deliveryAddress;

    @NotEmpty(message = "Order items cannot be empty")
    @Valid
    private List<OrderItem> items;

    // Other fields with validation
}


/**
 * Response DTO with serialization config
 */
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Schema(description = "Order response with all details")
public class OrderResponse {

    @Schema(description = "Order ID", example = "ord_12345")
    private String id;

    @Schema(description = "External order ID", example = "OMS-2024-001")
```

```
    private String orderId;

    @Schema(description = "Customer ID", example = "cust_12345")
    private String customerId;

    @Schema(description = "Current order status", example = "CREATED")
    private OrderStatus status;

    @JsonFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss")
    @Schema(description = "Order creation timestamp", example =
"2024-01-15T10:30:00")
    private LocalDateTime createdAt;

    // Other fields
}

/**
 * Paginated Response Wrapper
 */
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Schema(description = "Paginated response wrapper")
public class PagedResponse<T> {

    @Schema(description = "List of items in current page")
    private List<T> content;

    @Schema(description = "Current page number (0-based)", example = "0")
    private int page;

    @Schema(description = "Number of items per page", example = "20")
    private int size;

    @JsonProperty("total_elements")
    @Schema(description = "Total number of items across all pages", example = "150")
    private long totalElements;

    @JsonProperty("total_pages")
    @Schema(description = "Total number of pages", example = "8")
    private int totalPages;

    @Schema(description = "Whether this is the first page", example = "true")
    private boolean first;

    @Schema(description = "Whether this is the last page", example = "false")
    private boolean last;
}
```

7. EXCEPTION HANDLING DESIGN

7.1 Global Exception Handler

```
@RestControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

    /**
     * Handle validation errors
     */
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ErrorResponse> handleValidationException(
            MethodArgumentNotValidException ex) {

        List<String> errors = ex.getBindingResult()
            .getFieldErrors()
            .stream()
            .map(error -> error.getField() + ": " + error.getDefaultMessage())
            .collect(Collectors.toList());

        ErrorResponse response = ErrorResponse.builder()
            .timestamp(LocalDateTime.now())
            .status(HttpStatus.BAD_REQUEST.value())
            .error("Validation Failed")
            .message("Request validation failed")
            .details(errors)
            .build();

        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(response);
    }

    /**
     * Handle business exceptions
     */
    @ExceptionHandler(OrderNotFoundException.class)
    public ResponseEntity<ErrorResponse>
handleOrderNotFound(OrderNotFoundException ex) {
        ErrorResponse response = ErrorResponse.builder()
            .timestamp(LocalDateTime.now())
            .status(HttpStatus.NOT_FOUND.value())
            .error("Order Not Found")
            .message(ex.getMessage())
            .build();

        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(response);
    }

    /**
```

```
     * Handle invalid state transitions
     */
    @ExceptionHandler(InvalidOrderStateException.class)
    public ResponseEntity<ErrorResponse>
handleInvalidState(InvalidOrderStateException ex) {
        ErrorResponse response = ErrorResponse.builder()
            .timestamp(LocalDateTime.now())
            .status(HttpStatus.CONFLICT.value())
            .error("Invalid Order State")
            .message(ex.getMessage())
            .build();

        return ResponseEntity.status(HttpStatus.CONFLICT).body(response);
    }

    /**
     * Handle all other exceptions
     */
    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGenericException(Exception ex) {
        log.error("Unhandled exception: ", ex);

        ErrorResponse response = ErrorResponse.builder()
            .timestamp(LocalDateTime.now())
            .status(HttpStatus.INTERNAL_SERVER_ERROR.value())
            .error("Internal Server Error")
            .message("An unexpected error occurred")
            .build();

        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(response);
    }
}
```

7.2 Custom Exceptions

```
/**
 * Domain-specific exceptions
 */
public class OrderNotFoundException extends RuntimeException {
    public OrderNotFoundException(String message) {
        super(message);
    }
}
public class DeliveryPartnerNotFoundException extends RuntimeException {
    public DeliveryPartnerNotFoundException(String message) {
        super(message);
    }
}
public class InvalidOrderStateException extends RuntimeException {
```

```
    public InvalidOrderStateException(String message) {
        super(message);
    }
}
public class SagaExecutionException extends RuntimeException {
    public SagaExecutionException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

8. CONFIGURATION DESIGN

8.1 Application Configuration

```
@Configuration
public class AppConfig {

    /**
     * Order Processing Configuration
     */
    @Bean
    public OrderProcessingTemplate
orderProcessor(@Qualifier("standardOrderProcessor")
                                        OrderProcessingTemplate processor) {
        return processor;
    }

    /**
     * Strategy Configuration
     */
    @Bean
    @Primary
    public OrderAssignmentStrategy defaultAssignmentStrategy() {
        return new CityBasedAssignmentStrategy();
    }

    /**
     * Observer Registration
     */
    @Bean
    public OrderStatusChangeObserver orderStatusObserver(List<OrderObserver>
observers) {
        OrderStatusChangeObserver observer = new OrderStatusChangeObserver();
        observers.forEach(observer::registerObserver);
        return observer;
    }

    /**
     * Validation Chain Configuration
     */
    @Bean
    public OrderValidationChain orderValidationChain(
            CustomerValidationHandler customerHandler,
            RestaurantValidationHandler restaurantHandler,
            ItemsValidationHandler itemsHandler) {

        return new OrderValidationChain(customerHandler, restaurantHandler,
itemsHandler);
    }
}
```

8.2 OpenAPI Configuration

```
@Configuration
public class OpenApiConfig {

    @Bean
    public OpenAPI orderManagementOpenAPI() {
        return new OpenAPI()
            .info(new Info()
                .title("Order Management Service API")
                .description("API for managing food and grocery delivery orders")
                .version("1.0.0")
                .contact(new Contact()
                    .name("Swiggy Tech Team")
                    .email("tech@swiggy.com")
                    .url("https://www.swiggy.com"))
                .license(new License()
                    .name("MIT License")
                    .url("https://opensource.org/licenses/MIT")))
            .externalDocs(new ExternalDocumentation()
                .description("Order Management Service Documentation")
                .url("https://docs.swiggy.com/order-service"))
            .addSecurityItem(new SecurityRequirement().addList("Bearer Authentication"))
            .components(new Components()
                .addSecuritySchemes("Bearer Authentication",
                    new SecurityScheme()
                        .name("Bearer Authentication")
                        .type(SecurityScheme.Type.HTTP)
                        .scheme("bearer")
                        .bearerFormat("JWT")));
    }
}
```

9. DATABASE DESIGN

9.1 Entity Relationship Diagram

9.2 Index Strategy

```
-- Performance-critical indexes
CREATE INDEX idx_order_status ON orders(status);
CREATE INDEX idx_order_customer_id ON orders(customer_id);
CREATE INDEX idx_order_restaurant_id ON orders(restaurant_id);
CREATE INDEX idx_order_created_at ON orders(created_at DESC);
CREATE INDEX idx_order_city_status ON orders(delivery_city, status);
CREATE INDEX idx_partner_city_status ON delivery_partners(current_city, status)
WHERE is_active = true;
CREATE UNIQUE INDEX idx_order_idempotency ON order_events(idempotency_key)
WHERE processed_at IS NULL;
```

10. TESTING STRATEGY

10.1 Test Pyramid

10.2 Unit Test Examples

```
/**
 * Domain Model Tests
 */
class OrderTest {

    @Test
    void givenCreatedOrder_whenCancel_thenSuccess() {
        // Given
        Order order = OrderFactory.createOrder(...);

        // When
        order.cancel("Customer requested");

        // Then
        assertThat(order.getStatus()).isEqualTo(OrderStatus.CANCELLED);
        assertThat(order.getCancellationReason()).isEqualTo("Customer requested");
        assertThat(order.getCancelledAt()).isNotNull();
    }

    @Test
    void givenDeliveredOrder_whenCancel_thenThrowException() {
        // Given
        Order order = OrderFactory.createOrder(...);
        order.updateStatus(OrderStatus.DELIVERED);

        // When/Then
        assertThrows(IllegalStateException.class,
            () -> order.cancel("Too late"));
    }
}


/**
 * Service Layer Tests
 */
@ExtendWith(MockitoExtension.class)
class OrderServiceTest {

    @Mock private OrderRepository orderRepository;
    @Mock private OrderFactory orderFactory;
    @Mock private OrderValidationChain validationChain;
    @Mock private OrderStatusChangeObserver observer;
    @Mock private OrderCreationSaga saga;
```

```
    @InjectMocks private OrderService orderService;

    @Test
    void givenValidRequest_whenCreateOrder_thenSuccess() {
        // Given
        CreateOrderRequest request = CreateOrderRequest.builder()
            .customerId("cust123")
            .customerName("John Doe")
            .build();

        Order order = Order.builder().orderId("OMS-123").build();

        when(orderFactory.createOrder(any())).thenReturn(order);
        when(validationChain.validateOrder(any())).thenReturn(true);
        when(orderRepository.save(any())).thenReturn(order);

        // When
        Order result = orderService.createOrder(request);

        // Then
        assertThat(result.getOrderId()).isEqualTo("OMS-123");
        verify(saga).execute(order);
        verify(observer).notifyOrderCreated(order);
    }
}


/**
 * Strategy Pattern Tests
 */
class CityBasedAssignmentStrategyTest {

    @Test
    void givenSameCityPartners_whenSelect_thenChooseLeastLoaded() {
        // Given
        Order order = createOrderWithCity("Bangalore");
        DeliveryPartner partner1 = createPartner("Bangalore", 1); // 1 order
        DeliveryPartner partner2 = createPartner("Bangalore", 2); // 2 orders
        List<DeliveryPartner> partners = List.of(partner1, partner2);

        CityBasedAssignmentStrategy strategy = new CityBasedAssignmentStrategy();

        // When
        DeliveryPartner selected = strategy.selectDeliveryPartner(order, partners);

        // Then
        assertThat(selected).isEqualTo(partner1); // Least loaded
    }
}
```

11. DEPLOYMENT & SCALING

11.1 Containerization

```
# Dockerfile
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY target/order-management-service.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

11.2 Kubernetes Deployment

```
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: order-service
  template:
    metadata:
      labels:
        app: order-service
    spec:
      containers:
      - name: order-service
        image: order-service:1.0.0
        ports:
        - containerPort: 8080
        env:
        - name: SPRING_PROFILES_ACTIVE
          value: "production"
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: db.host
        resources:
          requests:
            memory: "512Mi"
            cpu: "250m"
          limits:
            memory: "1Gi"
            cpu: "500m"
        livenessProbe:
          httpGet:
            path: /actuator/health
```

```
          port: 8080
        initialDelaySeconds: 60
        periodSeconds: 10
      readinessProbe:
        httpGet:
          path: /actuator/health/readiness
          port: 8080
        initialDelaySeconds: 30
        periodSeconds: 5
```

11.3 Scaling Strategy

```
# Horizontal Pod Autoscaler
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: order-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: order-service
  minReplicas: 3
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 300
      policies:
      - type: Percent
        value: 50
        periodSeconds: 60
    scaleUp:
      stabilizationWindowSeconds: 60
      policies:
      - type: Percent
        value: 100
        periodSeconds: 30
```

12. MONITORING & OBSERVABILITY

12.1 Metrics Collection

```
@Component
@Slf4j
public class OrderMetrics {

    private final MeterRegistry meterRegistry;

    // Custom metrics
    private final Counter orderCreationCounter;
    private final Timer orderProcessingTimer;
    private final DistributionSummary orderAmountSummary;

    public OrderMetrics(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;

        this.orderCreationCounter = Counter.builder("order.creation.total")
            .description("Total orders created")
            .tag("service", "order-management")
            .register(meterRegistry);

        this.orderProcessingTimer = Timer.builder("order.processing.time")
            .description("Time taken to process order")
            .tag("service", "order-management")
            .register(meterRegistry);

        this.orderAmountSummary =
DistributionSummary.builder("order.amount.distribution")
            .description("Distribution of order amounts")
            .baseUnit("INR")
            .register(meterRegistry);
    }

    public void recordOrderCreation(Order order) {
        orderCreationCounter.increment();
        orderAmountSummary.record(order.getGrandTotal().doubleValue());

        // Tag by order type
        meterRegistry.counter("order.creation.by.type",
            "order_type", order.getOrderType().name(),
            "city", order.getDeliveryAddress().getCity())
            .increment();
    }
}
```

12.2 Health Indicators

```
@Component
public class OrderServiceHealthIndicator implements HealthIndicator {

    private final OrderRepository orderRepository;
    private final DatabaseHealthChecker dbHealthChecker;

    @Override
    public Health health() {
        // Check database connectivity
        if (!dbHealthChecker.isHealthy()) {
            return Health.down()
                .withDetail("database", "unavailable")
                .build();
        }

        // Check recent order creation
        try {
            long recentOrders =
orderRepository.countRecentOrders(LocalDateTime.now().minusMinutes(5));
            Health.Builder builder = Health.up()
                .withDetail("database", "connected")
                .withDetail("recent_orders", recentOrders);

            if (recentOrders == 0) {
                builder.withDetail("warning", "No orders in last 5 minutes");
            }

            return builder.build();

        } catch (Exception e) {
            return Health.down()
                .withDetail("database", "error")
                .withException(e)
                .build();
        }
    }
}
```

13. SECURITY DESIGN

13.1 JWT Authentication

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig {

    private final JwtAuthenticationFilter jwtAuthFilter;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/v1/health/**").permitAll()
                .requestMatchers("/swagger-ui/**", "/v3/api-docs/**").permitAll()
                .requestMatchers(HttpMethod.GET, "/api/v1/orders/**").hasRole("USER")
                .requestMatchers(HttpMethod.POST, "/api/v1/orders").hasRole("USER")
                .requestMatchers(HttpMethod.PUT,
"/api/v1/orders/*/status").hasRole("DELIVERY_PARTNER")
                .requestMatchers("/api/v1/delivery-partners/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )
            .authenticationProvider(authenticationProvider())
            .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration
config)
            throws Exception {
        return config.getAuthenticationManager();
    }
}
```

13.2 Idempotency Implementation

```
@Component
public class IdempotencyService {

    private final CacheManager cacheManager;

    /**
     * Idempotency Key Pattern:
     * - Client sends unique key in header
     * - Server checks if key already processed
     * - If processed, returns previous response
     * - If not, processes and stores result
     */
    public <T> T executeWithIdempotency(String idempotencyKey,
                                        Supplier<T> operation,
                                        Class<T> responseType) {

        // Check cache for existing result
        Cache cache = cacheManager.getCache("idempotency-cache");
        Cache.ValueWrapper cached = cache.get(idempotencyKey);

        if (cached != null) {
            log.info("Idempotency key {} already processed", idempotencyKey);
            return responseType.cast(cached.get());
        }

        // Execute operation
        T result = operation.get();

        // Cache result with TTL
        cache.put(idempotencyKey, result);

        return result;
    }
}
```

14. PERFORMANCE OPTIMIZATIONS

14.1 Caching Strategy

```
@Configuration
@EnableCaching
public class CacheConfig {

    @Bean
    public CacheManager cacheManager() {
        CaffeineCacheManager cacheManager = new CaffeineCacheManager();
        cacheManager.setCaffeine(Caffeine.newBuilder()
            .maximumSize(1000)
            .expireAfterWrite(5, TimeUnit.MINUTES)
            .recordStats());

        return cacheManager;
    }
}
@Service
@Slf4j
public class OrderService {

    @Cacheable(value = "orders", key = "#orderId")
    public Order getOrder(String orderId) {
        log.debug("Cache miss for order: {}", orderId);
        return orderRepository.findByOrderId(orderId)
            .orElseThrow(() -> new OrderNotFoundException(orderId));
    }

    @CacheEvict(value = "orders", key = "#order.orderId")
    public Order updateOrder(Order order) {
        return orderRepository.save(order);
    }
}
```

14.2 Database Optimization

```
@Repository
public interface OrderJpaRepository extends JpaRepository<OrderEntity, String> {

    /**
     * Use projection for read-only queries
     */
    @Query("SELECT new com.swiggy.order.dto.projection.OrderSummary(o.id,
o.orderId, " +
           "o.status, o.customerName, o.grandTotal, o.createdAt) " +
           "FROM OrderEntity o WHERE o.customerId = :customerId")
    Page<OrderSummary> findOrderSummariesByCustomerId(@Param("customerId")
String customerId,
                                                      Pageable pageable);

    /**
     * Batch fetching for N+1 problem
     */
    @EntityGraph(attributePaths = {"items", "deliveryAddress"})
    @Query("SELECT o FROM OrderEntity o WHERE o.id IN :ids")
    List<OrderEntity> findOrdersWithDetails(@Param("ids") List<String> ids);
}
```

15. FUTURE ENHANCEMENTS ROADMAP

15.1 Phase 1 (Current)

- ✅ Clean Architecture with Design Patterns
- ✅ Basic Order Lifecycle
- ✅ Delivery Partner Assignment
- ✅ REST APIs with Swagger

15.2 Phase 2 (Next 3 months)

- 🔄 Async Event Processing with Kafka
- 🔄 Real-time Tracking with WebSockets
- 🔄 Advanced Analytics Dashboard
- 🔄 A/B Testing Framework for Strategies

15.3 Phase 3 (Next 6 months)

- 🔄 Microservices Decomposition
- 🔄 Machine Learning for ETA Prediction
- 🔄 Dynamic Pricing Engine
- 🔄 Multi-language Support

15.4 Phase 4 (Next 12 months)

- 🔄 International Expansion
- 🔄 Voice-based Ordering
- 🔄 AR-based Delivery Tracking
- 🔄 Blockchain for Supply Chain

16. SUCCESS METRICS

16.1 Technical Metrics

- Availability: 99.9% uptime
- P99 Latency: < 500ms for order creation
- Error Rate: < 0.1% of requests
- Test Coverage: > 80% unit tests
- Deployment Frequency: Multiple times per day

16.2 Business Metrics

- Order Volume: 10K orders/minute peak
- Assignment Time: < 30 seconds average
- Delivery Time Accuracy: > 90% within ETA
- Customer Satisfaction: > 4.5/5 rating
- Partner Utilization: > 70% active time

SUMMARY

This LLD provides a comprehensive design for a production-ready Order Management Service that:

1. Follows Clean Architecture with clear separation of concerns
2. Implements 8 Design Patterns solving specific business problems
3. Scales horizontally with containerization and Kubernetes
4. Ensures reliability through Saga pattern and idempotency
5. Maintains high code quality with comprehensive testing
6. Provides observability through metrics and monitoring
7. Secures APIs with JWT authentication and authorization
8. Optimizes performance with caching and database tuning